-10

CS 354: Machine Organization and Programming **Spring 2016**

Lecture 8-10: Data Representation

Instructor: Shaleen Deep 2/12/2016

Before we start, let us review some basic notation.

As we know, computers understand only $0's$ and $1's$. Naturally, all characters and digits that we understand as humans, need to be translated to a bit pattern consisting of 0 and 1. This representation is called *binary representation*.

Binary representation works on arithmetic with base 2. Note that there are other representations also possible with a different base. The most common that we use is the *decimal representation* that uses base 10. Other common representations are called *Octal* (denoted using a 0 before the number) and *Hexadecimal* (denoted by putting $0x$ before the number). For example, 027 is the *octal* representation for the decimal number 17 and $0x11$ is the corresponding *hexadecimal* representation.

8.1 Why Data Representation

Without applying any semantics to bit patterns, they are nothing more than a sequence of $0's$ and $1's$.

Example 8.1. *Consider the bit pattern 1000001. If this is a character, it's value is 'A', but interpreting this as an integer gives us a value of* 65.

In order to apply this interpretation, languages provide us with *data types* to apply semantics to binary numbers. For example, *C language* provides us with data types such as *int, float, double, char* etc.

8.1.1 Word Size

Formally, *word size* refers to number of bits processed by a CPU. In other words, this is the size of the registers of the CPU. Note that this definition is same as the one we discussed in class where we said that word size refers to the number of bits required to refer an address in the memory. Since the *instruction pointer*, which stores the address of the next instruction, is also a register in the CPU, the two definitions are the same.

Most modern systems have a *word size* of 32 or 64 bit. This means that a 32-bit system can refer to 2^{32} memory locations and a 64-bit system can refer to 2^{64} memory locations.

8.2 Endianess of a system

Before we understand *endinaness* and why it exists, we should note that memory is always *byte addressable*. What this means that the smallest possible independent unit that we can access from memory is a *byte*. Another way to look at this is that every byte of data has a unique memory address.

Example 8.2. *Consider the memory snippet below*

10000000	*10000000*	*10000000*	*10000000*

The first byte contains 10000000 *which means that the CPU has to read all the bits at this particular address. It cannot read, say, the first four bits only. The CPU can read more than 1 byte if it wants (which is what happens for types such as int which are 4 bytes long). Also, note that the convention(for a single byte) is to have the* most significant bit *on the leftmost end (which is* 1 *in our example) and the* least significant bit *is on the rightmost end (which is* 0 *in our example). This does not change (ever).*

Equipped with this fact, we immediately notice the problem. For data types that take more than 1 byte, there are multiple possible ways to order them.

Example 8.3. *Consider the number* $0x0A0B0C0D$*. Note that* $0A$ *is the most significant byte and* $0D$ *is the least significant byte. We can choose to represent this multi byte number in 2 of the following ways. Assume that the addresses of the memory increase from left to right.*

0A	*0B*	*0C*	*0D*

The representation above is called the Big Endian *where the* Most significant byte *comes at the lowest address.*

0D	*0C*	*0B*	*0A*

The representation above is called the Little Endian *where the* Least significant byte *comes at the lowest address.*

The little-endian system has the property that the same value can be read from memory at different lengths without using different addresses (even when alignment restrictions are imposed). For example, a 32-bit memory location with content $4A000000$ can be read at the same address as either 8-bit (value = $4A$), 16-bit ($004A$), 24-bit ($00004A$), or 32-bit ($0000004A$), all of which retain the same numeric value. On the other hand, big-endian systems have the advantage of ease of debugging for programmers since the logs and memory dumps contain the values in the *natural order* where the *most significant byte* is on the left.

For all practical purposes, there is no difference between these two representations. As application developers, the compiler abstracts us from the low level details. The only two places where ordering maybe of interest to us are listed below

- **System programming** - If you are writing a new compiler or device driver, you need to be aware of these low level details.
- **Network byte ordering** - If two systems want to communicate over network, it may be possible that they have different underlying architectures. Therefore, as a convention, we define the *Network Byte Order* as big endian i.e all communication over the network will have data ordered in the big endian format.

8.3 Boolean Algebra

A rich body of mathematical literature has been built around $0's$ and $1's$. The most notable of these being Boolean Algebra, that defines the logic for a set of given operators. The most common operations include *AND* (denoted by $\wedge$), *OR* (denoted by $\vee$), *NOT* (denoted by $\neg$) and *XOR* (denoted by $\oplus$). The corresponding *C language* operators are &&, ||, ! and $\oplus$

Truth table for the operators is given below

P	Q	$P \vee Q$	$P \wedge Q$	$P \oplus Q$	$\neg P$
0	0	0	0	0	1
0	1	1	0	1	1
1	0	1	0	1	0
1	1	1	1	0	0

8.4 Bitwise Operators

In the C programming language, operations can be performed on a bit level using bitwise operators. C provides six operators for bit manipulation namely *AND* (denoted by &), *OR* (denoted by |), *XOR* (denoted by $\wedge$), *left shift* (denoted by «), *right shift* (denoted by ») and *NOT* (denoted by ~)

The bitwise operators AND, OR, NOT and XOR behave in the same ways the logical operators except that bitwise operators operate on the bit value rather than the truth value.

The symbol of right shift operator is ». For its operation, it requires two operands. It shifts each bit in its left operand to the right. The number following the operator decides the number of places the bits are shifted (i.e. the right operand). Thus by doing ch » 3 all the bits will be shifted to the right by three places and so on.

Example 8.4. *If the variable ch contains the bit pattern* 11100101*, then ch »* 1 *will produce the result* 01110010*, and ch »* 2 *will produce* 00111001

Here blank spaces are generated simultaneously on the left when the bits are shifted to the right. When performed on an unsigned type, the operation performed is a logical shift, causing the blanks to be filled by $0's$. When performed on a signed type, the result is technically undefined and compiler dependant, however most compilers will perform an arithmetic shift, causing the blank to be filled with the sign bit of the left operand.

The symbol of left shift operator is «. It shifts each bit in its left-hand operand to the left by the number of positions indicated by the right-hand operand. It works opposite to that of right shift operator. Thus by doing ch « 1 in the above example we have 11001010. Blank spaces generated are filled up by $0's$ as above.

8.5 Unsigned and signed integers

In order to represent *unsigned integers*, the most significant digit is used to denote whether the number is positive or negative.

Example 8.5. *If we represent using* 4 *bits, then* $+5$ *is represented as* 0101 *and* -5 *is represented as* 1011 *(this is in* $2's$ *complement).*

8.6 2's complement

In order to represent negative numbers, we have 3 possible representations

- Signed magnitude form - The negative of a number is represented by just flipping the sign bit. For example, -5 is represented 1101. The MSB tells us that the number is negative and 101 is the representation of 5.
- 1's complement - We define the $1's$ complement a bit representation as inversion of all the bits. For example, the $1's$ complement of 101 is 010. We represent -5 as 1010 where the MSB tells us that the number is negative and 010 is the $1's$ complement of 5. The problem with this representation is that 0 can be represented as 0000 and 1111. (*Do you see why?*)
- 2's complement - We define the $2's$ complement of a bit representation as 1 + $1's$ complement of the number. In this representation, we represent -5 as 1011 where 011 is the *2's complement* of 5 (because $1's$ complement is 010 and $010 + 1$ is 011)

Negative numbers are represented using $2's$ complement in the memory.

8.7 Integer Arithmetic

8.7.1 Unsigned Addition

Consider two number $0 \leq x, y \leq 2^w - 1$ where w is the size of our integer representation. It is easy to see that $0 \leq x + y \leq 2^{w+1} - 2$. Note that the sum may require $w + 1$ bits. Therefore, there maybe cases when we need to *truncate* a bit from the result. Such a condition is called *overflow*.

Example 8.6. *Let $w = 4$. Consider $x = 10$ and $y = 10$. Then, $x + y = 20$. In binary representation, $x + y = 10100$. But since we have only 4 bits to store the output, we truncate the MSB (which is 1) and the result is 0100 which is 4. Note that $20\%16 = 4$.*

In general, whenever an overflow occurs, the result is modulo 2^w (*You should try a few more example to see this. Can you see why this is the case? Hint - What is the weight of the MSB that is dropped?*)

8.7.2 Signed Addition

Signed addition is a little more complicated in that we have to deal with overflow on the negative side as well. As before, Consider two number $-2^{w-1} \leq x, y \leq 2^{w+1} - 1$ where w is the size of our integer representation. It is easy to see that $0 \leq x + y \leq 2^w - 2$. Note that we need $w + 1$ bits to represent these numbers (remember that 1 bit is reserved for the sign bit).

Example 8.7. *The positive overflow case is the same as unsigned addition. Let $w = 4$. To see a negative case, let $x = -8$ and $y = -5$, then $x + y = -13$. x and y in $2'$s complement are 1000 and 1011 respectively. $x + y$ is 10011 but we need to truncate the MSB. Therefore, the result is 0011 which 3. Note that our answer is $2^4 + -13 = 3$. For the positive case, sum $= x + y - 2^w$. You can try adding $5 + 5$ to verify this.*

8.8 Multiplication

Signed and unsigned multiplication are also modulo 2^w in nature, that is, if the result *overflows* and *underflows*, the result is modulo 2^w.

Exercise 8.8. *Multiply -3 and 3. Verify that the truncated result is -1.*

8.8.1 Multiplication by constants

Multiplication by constants is an optimization that a compiler makes when it knows at compile time the values being multiplied.

Example 8.9. *Let int a = 14*b. We can represent* 14 *as* $2^3 + 2^2 + 2^1$ *and consequently, int a = b«3 + b«2 + b«1. This converts a complex multiplication operation into 3 efficient bitwise operations. Note that we can do the multiplication in steps as int a = b«4 - b«1. We will see in a minute how shift operators work.*

8.9 Using shift operators

Ww know that « and » operators are used to manipulate bit patters by shifting. Dividing/Multiplying by a power of 2 can also be performed using shift operations, but we use a right shift rather than a left shift.

Example 8.10. ***Left shift*** *- A left shift of a bit pattern adds* $0's$ *to the right hand side. In effect, one left shift operation multiplies the number by 2. For example,* $2 << 1 = 4$ *because* 2 *in binary is* 0010 *and* 4 *is* 0100*. Note that if we do left shift enough number of times, the number would eventually become* 0*.*
Right shift *- Assuming the right shift is logical, A right shift of a bit pattern adds* $0's$ *to the left hand side. In effect, one right shift operation divides the number by 2. For example,* $2 >> 1 = 1$ *because* 2 *in binary is* 0010 *and* 1 *is* 0001*. Note that if we do left shift enough number of times, the number would eventually become* 0*.*

Exercise 8.11.

- *Find if a number is a power of 2*
- *Find if a number is even or odd*
- *Find if* i^{th} *bit in a number is 1 or 0*
- *Count total number of set bits in a number*
- *Detect if 2 numbers have opposite signs (Hint : What does* $x \wedge y$ *do?)*

8.10 Floating point numbers

Floating point numbers contain 2 parts - the fractional part and the integral part. Floating point numbers are stored in a format known as *IEEE-754*. In general, floats have a size of 32 bits and double have a size of 64 bits on most machines.

The biggest issue that we face in representing fractional numbers is that some fractions do not have a finite representation

Example 8.12. *Consider the number 0.9 in decimal. The binary representation is 0.1 [1100][1100][1100].. Since the float type has only 32 bits, we have to sacrifice some precision. Therefore, after storing 0.9 in the float variable, it's value will not be exactly 0.9*

IEEE floating point numbers have three basic components: the sign, the exponent, and the mantissa. The mantissa is composed of the fraction and an implicit leading digit. Laid out as bits, single floating point numbers look like this - SEEEEEEEE MMMMMMMMMMMMMMMMM-MMMMMM where S is for 1 sign bit, E is for 8 exponent bits and M is for 23 mantissa bits. The exponent field needs to represent both positive and negative exponents. To do this, a bias is added to the actual exponent in order to get the stored exponent. For IEEE single-precision floats, this value is 127. Thus, an exponent of zero means that 127 is stored in the exponent field. A stored value of 200 indicates an exponent of $(200 - 27)$, or 73.

The key takeaway point is to note that due to loss of precision, using == and != operators on floating point numbers is not a good idea. We should always check the difference between 2 floating numbers to determine if 2 floats are very close to each other (For example, if for your purpose a resolution of 0.0001 is sufficient, you should check if the difference between floating variable and (say) $0.9f$ is less than 0.0001).